Санкт-Петербургский государственный университет

Кафедра Системного программирования

Группа 21.Б10-мм

Miminet

Минязев Роман Русланович

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
старший преподаватель кафедры системного программирования,
Зеленчук Илья Валерьевич

Санкт-Петербург
2024

Оглавление

Введение

С постоянным развитием компьютерных сетей, вырастает потребность в специалистах в этой области, в связи с чем во многих технических вузах проводятся курсы, в рамках которых изучаются основы компьютерных сетей. Например, курс «Компьютерные сети и хранилища данных» на математико-механическом факультете или «Практикум по сетевым технологиям» на физическом факультете СПБГУ.

Однако, теоретические части многих курсов являются довольно абстрактными, в связи с чем многие студенты сталкиваются с непониманием материала. Кроме того, визуальная информация воспринимается гораздо проще чем сырой текст. Например, простейшая анимация, демонстрирующая процесс установления TCP-сеанса между клиентом и сервером, позволит обучающемуся быстрее запомнить возможные значения флагов в TCP-заголовках.

Так или иначе, для подкрепления теоретических знаний, обучающимся необходимо научиться проектировать сети, в связи с чем появляется необходимость использования сетевых эмуляторов.

Несмотря на текущее обилие сетевых эмуляторов, их использование уже подразумевает наличие определенных знаний о сетях и устройстве эмулятора, а также требует от пользователя определенное время на установку нужного программного обеспечения и т.д. Более того, один из самых известных эмуляторов Cisco Packet Tracer недоступен в Россиийской Федерации.

Miminet — веб-эмулятор компьютерных сетей для образовательных целей. Используя сетевые компоненты(свитчи, хабы, ...), пользователь получает возможность проектировать различные сети, которые впоследствии эмулируются на серверах с операционными системами из семейства GNU/Linux.

Так как серверная часть Miminet не является кроссплатформенной, а также имеет достаточно много зависимостей, число которых при добавлении новой функциональности будет только увеличиваться, новые разработчики в команде будут тратить достаточно большое коли-

чество времени на развертывание приложения. Более того, frontend-разработчики, которые не участвуют в разработке backend-части приложения должны иметь возможность в достаточной степени абстрагироваться от требований серверной части и вести разработку в привычных им инструментах и операционных системах.

Кроме того, пропускная способность приложения является довольно низкой, так как на эмуляцию простейших сетей тратится около 7-10 секунд, а на эмуляцию более сложных сетей может потребоваться достаточно большое количество времени(например, при использовании протокола STP для устранения петель в сетевых топологиях на эмуляцию будет тратиться больше 30 секунд). Данную проблему изначально нельзя было решить горизонтальным масштабированием, так как у Miminet обладал монолитной архитектурой. Более того, вертикальное масштабирование не сильно бы изменило ситуацию, так как в текущей реализации сети обрабатываются последовательно.

1. Постановка задачи

Целью работы является автоматизация создания сред для разработки, изменение архитектуры приложения на масштабируемую с последующим рефакторингом кода, офоромление репозитория с настройкой CI/CD.

Для её выполнения были поставлены следующие задачи:

1. Провести сравнительный обзор технологий.
2. Спроектировать и реализовать масштабируемую архитектуру.
3. Конфигурация среды на операционных системах, отличных от Debian, Fedora GNU/Linux.
4. Провести рефакторинг кода.
5. Написать модульные тесты.
6. Настройка CI/CD.

2. Обзор

2.1. Выбор брокера сообщений

Для построения распределенной системы и реализации паттерна издатель/подписчик, а также маршрутизации сообщений было необходимо выбрать брокер сообщений. Среди рассматриваемых брокеров были RabbitMQ и Redis. Несмотря на то что операции с Redis[1] происходят непосредственно в памяти, что делает его куда более производительным решением, позволяя обрабатывать большее количество запросов, RabbitMQ превосходит Redis в других показателях.

RabbitMQ[2] — специализированный брокер сообщений, реализующий протокол AMQP[3], который вводит понятия Routing Key, Exchange, Queue, Binding, в следствие чего обладает более продвинутой маршрутизацией, чем Redis [5]. Redis поддерживает стандартный механизм pub/sub и маршрутизацию с помощью сопоставления по шаблону, в то время как RabbitMQ [4] имеет множество типов Exchange (Direct, Fanout, Topic, Headers, Custom), вводит Exchange to Exchange binding и т.д..

Также RabbitMQ даёт определенные гарантии на доставку сообщений. Сообщения будут надежно храниться в очередях до момента получения подтверждения успешной обработки сообщения и в случае сбоев ждать доступности потребителей. Но также существует возможность удалять сообщения из очереди сразу после его доставки в приложение для оптимизации использования ресурсов и более высокой пропускной способности. Очереди могут быть постоянными, временными или автоматически удаляемыми.

2.2. Выбор хранилища для результатов задач

Необходимо выбрать хранилище для временного хранения результатов эмуляций и метаданных для обеспечения отслеживания состояния

[1]Redis: https://redis.io/ (дата обращения: 15.12.2023)
[2]RabbitMQ: https://www.rabbitmq.com/ (дата обращения: 15.12.2023)
[3]AMQP: https://www.amqp.org/ (дата обращения: 15.12.2023)

задач и асинхронного получения их результатов. В качестве хранилища можно использовать RabbitMQ с механизмом RPC. При отправке сообщения клиент может создать долговременную очередь для ответов и подписаться на нее. После отправки сообщения с заголовком reply-to результаты отправляются в нужную очередь. Однако, данный процесс можно оптимизировать, используя механизм direct reply-to, подписавшись на виртуальную очередь amqp.rabbitmq.reply-to (сообщения не будут храниться на диске и не будет создан отдельный Erlang процесс). Вследствие чего необходимо автоматическое подтверждение сообщений, что может быть не самым безопасным решением. Также RPC вызов является блокирующим (в контексте direct reply-to для отправки сообщений и подписки на виртуальную очередь будет использоваться одно и то же соединение и канал), чего можно избежать, открыв новые каналы, но потратив при этом ресурсы. Учитывая все эти проблемы, для хранения результатов используется Redis, который является более производительным решением.

2.3. Celery

Celery[4] — распределённая очередь задач для асинхронной обработки сообщений.

Celery выбран по следующим причинам:

1. Celery расширяет позволяет применять определенные политики в отношении задач(количество повторных попыток, временные промежутки для повторного выполнения задач, отложенные задачи, повторная попытка при определенных исключениях и т.д.).

2. Настройка многопроцессной обработки на стороне worker'ов настраивается несколькими параметрами, без необходимости написания кода.

3. Не нужно писать логику для распределения задач на стороне worker. Вызов задач напоминает RPC (но неблокирующий, кли-

[4]Celery: https://github.com/celery/celery (дата обращения: 15.12.2023)

ент не ожидает, пока сервер закончит обработку, а вместо этого отслеживает состояние задачи).

4. Кэширование при обращении к бэкенду для оптимизации процесса отслеживания состоянии задачи.

5. Настройка стратегий для обработки отказов одного из RabbitMQ узлов (подходящая стратегия по умолчанию — round-robin)

2.4. Vagrant

Vagrant[5] [2] — менеджер виртуальных машин, используемый для создания и управлением средами разработки.

Mininet[6] [3] требует root прав для доступа к сетевым интерфейсам хоста, arp-tables и т.д. Очевидно, что необходима изоляция системы и Mininet с IPMininet[7] лучше использовать в виртуальных машинах.

Более того, frontend-часть Miminet является кроссплатформенной, в то время как для Mininet требуется Debian, Fedora GNU/Linux. Также, Mininet и IPMininet могут иметь большое количество зависимостей, в связи с чем у frontend-разработчиков на других операционных системах или у новых разработчиков в команде могут возникнут некоторые трудности, так как на настройку виртуальной машины и скачивание всех зависимостей самостоятельно может уйти достаточно большое количество времени. Например, при использовании shared folder в VirtualBox[8], vboxfs даже не в состоянии справиться с pip install -r, в связи с чем используется SMB[9] для Windows, macOS и NFS[10] для Debian, Fedora. Эти проблемы решаются при помощи Vagrant.

[5]Vagrant: https://www.vagrantup.com/ (дата обращения: 15.12.2023)
[6]Mininet: https://github.com/mininet/mininet (дата обращения: 15.12.2023)
[7]IPMininet: https://github.com/cnp3/ipmininet (дата обращения: 15.12.2023)
[8]VirtualBox: https://www.virtualbox.org/ (дата обращения: 15.12.2023)
[9]SMB: https://en.wikipedia.org/wiki/Server_Message_Block (дата обращения: 15.12.2023)
[10]NFS: https://datatracker.ietf.org/doc/html/rfc3010 дата обращения: 15.12.2023

2.5. Docker

Использование Docker[11] [1] без виртуальных машин является не самым безопасным решением, так как Mininet требует многие привилегии (net_admin, net_raw, ...), однако, это более дешёвое решение с точки зрения используемых ресурсов, которое подходит для развёртывания на машинах (которые можно легко и дешёво заменить в случае выхода из строя) с минимальной конфигурацией, которой хватит под работу worker'а. Также, несмотря на определенные риски, это является куда более удобным и автоматизированным решением для локального развертывания.

[11]Docker Compose: https://github.com/docker/compose (дата обращения: 15.12.2023)

3. Реализация

3.1. Rabbitmq

3.1.1. x-consistent-hash

В проекте используется x-consistent-hash[12] плагин, который позволяет равномерно распределять сообщения по очередям на основании Routing Key (в зависимости от веса привязки). Это обеспечивает равномерное распределение в случае изменения топологии (добавлении новых очередей) и позволяет абстрагироваться приложению от информации о очередях и их привязках, которая была бы необходима для организации распределения сообщений по очередям.

3.1.2. Кластеризация

RabbitMQ имеет простой процесс кластеризации. Для добавления узла в кластер и обеспечения высокой доступности нужно синхронизировать .erlang.cookie на всех узлах и применить определенные политики (на данный момент это делается простейшим образом, с помощью аргумента all происходит репликация на все узлы кластера).

При отказе одного из RabbitMQ узлов, Celery (на основе фреймворка Kombu) предпринимает определенное количество попыток с экспоненциальным откатом для попытки повторного установления соединения и в случае неудачи устанавливает соединение с другим узлом по стратегии round-robin.

3.2. Backend

Celery забирает сообщения из нужных очередей, после чего начинается процесс десериализации JSON сети. Далее, создаются все сетевые устройства и интерфейсы, происходит выдача ip адресов, настройка шлюзов по умолчанию и т.д. После того как сеть полностью сгенериро-

[12]x-consistent-hash: https://github.com/rabbitmq/rabbitmq-consistent-hash-exchange (дата обращения: 15.12.2023)

вана, на устройствах выполняются определенные команды. Например, nc -uq1 для отправления данных по протоколу UDP с хоста на сервер. Во время эмуляции происходит захват сетевого трафика при помощи tcpdump, на основании чего строится анимация, которую видит пользователь.

В ходе работы был отрефакторен бекэнд и написаны модульные тесты при помощью фреймворка pytest[13].

3.3. Локальное развертывание

Для локального развертывания были написаны конфигурационные файлы для Docker Compose, использующие Rabbitmq, Redis и Nginx образы. Так как Mininet требует root прав для доступа к определенным сетевым устройствам и интерфейсам, используется privileged флаг и network_mode: host. Однако, время эмуляций при использовании Docker Compose относительно времени эмуляций в виртуальной машине (без последующей контееризации с Docker) увеличивается пропорционально сложности сети (несмотря на network_mode: host), что не является особо критичным фактором для локального развертывания или staging, но является таковым для развертывания на production серверах.

Более того, для работы Docker необходимо установить в ядро модуль br_netfilter, вследствие чего для каждого linux-bridge будут использоваться правила для фильтрации и обработки трафика. Для ожидаемого поведения при эмуляции необходимо выставить значения параметров net.bridge.bridge-nf-call-iptables и net.bridge.bridge-nf-call-arptables в 0.

3.4. CD

На серверах создается пользователь, после чего на хост монтируется public key с флагами no-port-forwarding, no-X11-forwarding, no-agent-forwarding, no-pty, command. При подключении по ssh пользователю разрешено запускать только скрипт для развертывания, запрещено пе-

[13]Pytest: https://github.com/pytest-dev/pytest (дата обращения: 15.12.2023)

ренаправление портов, ввод иных команд и т.д.

3.5. Репозиторий

Был создан новый репозиторий для Miminet с Apache License 2.0. Настроены модульные тесты, линтер Flake8[14], форматтер Black[15], статическая проверка типов mypy[16] в CI. Также настроен Dependabot и CodeQL.

[14]Flake8 : https://github.com/PyCQA/flake8 (дата обращения: 15.12.2023)
[15]Black: https://github.com/psf/black (дата обращения: 15.12.2023)
[16]mypy: https://github.com/python/mypy (дата обращения: 15.12.2023)

4. Апробация

Для проверки корректности работы отдельных функций и модулей после рефакторинга, были написаны модульные тесты с использованием фреймворка pytest.

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты:

1. Проведен сравнительный обзор технологий.
2. Спроектирована и реализована масштабируемая архитектура.
3. Проведен рефакторинг кода.
4. Написаны модульные тесты.
5. Настроен CI/CD.

Ссылка на репозиторий: https://github.com/mimi-net/miminet.

Список литературы

[1] Docker. — URL: https://docs.docker.com/ (online; accessed: 2023-12-15).

[2] HashiCorp. — URL: https://developer.hashicorp.com/vagrant/docs (online; accessed: 2023-12-15).

[3] Mininet. — URL: https://github.com/mininet/mininet/wiki/Documentation (online; accessed: 2023-12-15).

[4] RabbitMQ. rabbitmq doc. — URL: https://www.rabbitmq.com/documentation.html (online; accessed: 2023-12-15).

[5] Redis. — URL: https://redis.io/docs/ (online; accessed: 2023-12-15).